FDTool: a Python application to mine for functional dependencies and candidate keys in tabular data

Matt Buranosky,
Elmar Stellnberger,
Emily Pfaff,
Cavin Ward-Caviness

June 13, 2018

Abstract

Functional dependencies and candidate keys crucially serve in the process of table decomposition, database normalization, and data cleansing. In this paper, we present FDTool, a command line Python application to discover functional dependencies in tabular datasets and infer candidate keys from them.

We are working out of **https://github.com/mburanosky17/FDTool**.

1 Functional Dependencies

Functional dependencies (FDs) are researched to better understand how attributes in a database schema relate to one another. An FD defines a rule constraint between two sets of attributes in a relation $r(U)$, where $U = \{v_1, v_2, \ldots, v_m\}$ is a finite set of attributes. A combination of attributes over a dataset is called a *candidate*. An FD $X \rightarrow Y$ asserts that the values of candidate X uniquely determine those of candidate Y. For example, the social security number (SSN) attribute in a dataset of public records functionally determines the first name attribute; we write $\{SSN\} \rightarrow \{first\_name\}$.

Definition 1.1. A *functional dependency* $X \rightarrow Y$, where $X, Y \subseteq U$, is satisfied by $r(U)$, if for all pairs of tuples $t_i, t_j \in r(U)$, we have that $t_i[X] = t_j[X]$ implies $t_i[Y] = t_j[Y]$.

In this case, X is the *left hand side* of an FD, and Y is the *right hand side*. If Y is not functionally dependent on any proper subset of X, then $X \rightarrow Y$ is *minimal*. Minimal FDs are our only concern in rule mining FDs, since all other FDs are logically implied. For instance, if we know $\{SSN\} \rightarrow \{first\_name\}$, then we can infer that $\{SSN,\ last\_name\} \rightarrow \{first\_name\}$.

1.1 Power set lattice

The search space for FDs is represented as a *power set lattice* of nonempty attribute combinations.

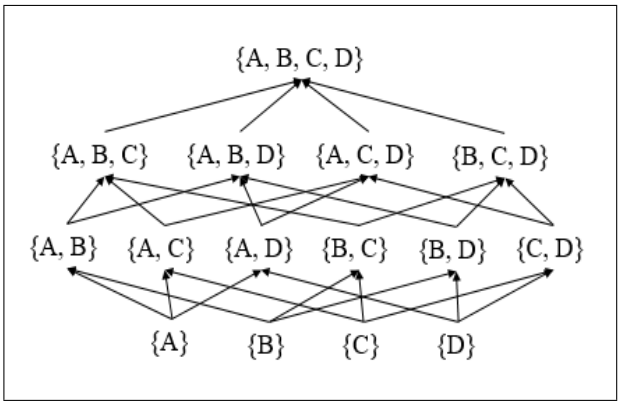

Figure 1: Nonempty combinations of attributes A, B, C, and D by k-level.

Figure 1 gives the nonempty combinations of attributes of a relation $r(U)$ such that $U = \{A, B, C, D\}$. There are $2^n - 1 = 2^4 - 1 = 15$ attribute subsets in the power set lattice. Each combination X of the attributes in U can be the left hand side of an FD $X \rightarrow Y$ such that $X \rightarrow Y$ is satisfied by relation $r(U)$. Since the attribute set itself U trivially determines every one of its subsets, it can be ignored as a candidate. There remain $2^n - 2 = 2^4 - 2 = 14$ nonempty, proper subsets of U that are to be considered candidates.

There are $n \cdot 2^{n-1} - n = 4 \cdot 2^{4-1} - 4 = 28$ edges (or arrows) in the semi-lattice of the complete search space for FDs in relation $r(U)$. The size of the search space for FDs is exponentially related to the number of attributes in U. Hence, the search space for FDs increases quite significantly with a greater number of

attributes in U. For instance, when there are 10 attributes in a relation, the search space for FDs climbs to 5110. This gives reason to be cautious of runtime and memory costs when deploying a rule mining algorithm to discover FDs.

1.2 Partition

The algorithms used to discover FDs differ in their approach to navigating the complete search space of a relation. Their candidate pruning methods vary and sometimes the methods used to validate FDs do as well. These discrepancies competitively affect runtime and memory behavior when used to process tables of different shapes.

The most common data structure used to validate FDs is the partition. A partition places tuples that have the same values on an attribute into the same group.

Definition 1.2. Let $X \subseteq U$ and let $t_1, \ldots, t_n$ be all the tuples in a relation $r(U)$. The *partition* over X, denoted $\prod_X$, is a set of the groups such that t_i and t_j, $1 \leq i, j \leq n$, $i \neq j$, are in the same group if and only if $t_i[X] = t_j[X]$.

It follows from Definition 1.2 that the *cardinality of the partition* $\mathbf{card}(\prod_A(r))$ is the number of groups in partition $\prod_A$. The cardinality of the partition offers a fast approach to validating FDs in a dataset.

Theorem 1.1. An FD $X \rightarrow Y$ is satisfied by a relation $r(U)$ if and only if $\mathbf{card}(\prod_X) = \mathbf{card}(\prod_{XY})$.

Theorem 1.1 provides an efficient method to check whether an FD $X \rightarrow Y$ holds in a relation. [1]

1.3 Closure

Efforts in relational database theory have lead to more runtime and memory efficient methods to check the complete search space of a relation for FDs. In place of needing each arrow in a semi-lattice checked, we can infer the FDs that logically follow from those already discovered. Such FDs are to be discovered as a consequence of *Armstrong's Axioms*, which are as follows: *Reflexivity*: $Y \subseteq X$ implies $X \rightarrow Y$; *Augmentation*: $X \rightarrow Y$ implies $XZ \rightarrow YZ$; *Transitivity* : $X \rightarrow Y$ and $Y \rightarrow Z$ imply $X \rightarrow Y$.

[1] FDTool uses Theorem 1.1 as means to check FDs in the GetFDs module with the *pandas* data analysis library functions *nunique()* and *dropduplicates.count()*.

These axioms signal the distinction between FDs that can be inferred from already discovered FDs, and those that cannot (Maier 1983). Exploiting Armstrong's Axioms allows us to avoid having to check as many candidates on the right hand side of FDs.

Definition 1.3. Let F be a set of functional dependencies over a dataset D and X be a candidate over D. The *closure of candidate* X with respect to F, denoted X^+, is defined as $\{Y \mid X \rightarrow Y$ can be deduced from F by Armstrong's Axioms$\}$.

The *nontrivial closure of candidate* X with respect to F is defined as $X^* = X^+ \setminus X$ and written X^*. [2]

2 Pruning Rules

Existing functional dependency algorithms are split between three categories: *Difference- and agree-set algorithms* (e.g., Dep-Miner, FastFDs), *Dependency induction algorithms* (e.g., FDEP), and *Lattice traversal algorithms* (e.g., TANE, FUN, FD_Mine, DFD).

2.1 Lattice traversal algorithms

Lattice traversal algorithms model the search space of a relation as a power set lattice. Most of such algorithms, i.e., TANE, FUN, and FD_Mine, use a level-wise approach to traversing the search space of a relation from the bottom-up. They start by checking for FDs that are singleton sets on the left hand side and iteratively transition to candidates of greater cardinality.

Definition 2.1. Let $X_1, X_2, \ldots, X_k, X_{k+1}$ be $(k+1)$ attributes over a database D. If $X_1X_2 \ldots X_k \rightarrow X_{k+1}$ is an FD with k attributes on its left hand side, then it is called a *k-level* FD.

The search space for FDs is reduced at the end of each iteration using pruning rules. *Pruning rules* check the validity of candidates not yet checked with FDs already discvored and those inferred from Armstrong's Axioms. After a search space is pruned, an *Apriori_Gen* principle generates k-level candidates with the

[2] FDTool saves the closure of candidates at each level before releasing it from memory at levels that follow.

$(k-1)$-level candidates that were not pruned. [3] Pruning reduces the number of candidates that are to be checked on the left hand side of FDs.

Apriori_Gen:

- *OneUp*: generates all possible candidates in C_k from those in C_{k-1}.
- *OneDown*: generates all possible candidates in C_{k-1} from those in C_k.

Level-wise lattice traversal algorithms stop iterating after all candidates in a search space are pruned. In this case, *Apriori_Gen* generates the null set $\emptyset$ raising a flag for the algorithm to terminate. This has the effect of shortening runtime to the degree that FDs are discovered and others are inferred.

The level-wise lattice traversal algorithms, TANE, FUN, and FD_Mine, differ in terms of pruning rules. TANE, for example, uses partitions to discover FDs. If the FD $AC \rightarrow B$ had been discovered, then $ACD \rightarrow B$ and $ACDE \rightarrow B$ are implied and need not be checked.

2.2 FD_Mine

FD_Mine performs favorably to many other of such algorithms on relational database schemas made up of long and narrow tables, i.e., those with many rows and few columns (≤ 14). This puts FD_Mine in a special position to rule mine bioinformatics data consisting of long sets of patient records.

Definition 2.2. Let X and Y be candidates over a dataset D. If $X \rightarrow Y$ and $Y \rightarrow X$ hold, then we say that X and Y are an *equivalence* and denote it as $X \leftrightarrow Y$.

Let R be a relational schema and X be a candidate of R over a dataset D. If $X \cup X^* = R$, then X is a *key*.

2.3 Non-minimal FDs

Prune(C_k, E, $Closure$) [4]

[3] *Apriori_Gen* methods differ in approach across implementations. FDTool computes the power set of remaining k-level candidates and collects the members who share a cardinality of $k+1$.

[4] Closure $= \{X^+ \mid X \in C_k \vee X \in OneDown\,[C_k]\}$.

```
01 for each S ∈ C_k:
02     for each X ∈ OneDown [C_k]:
03         if (X ⊂ S) then:
04             S^+ = S^+ ∪ X^*
05             if (X ∈ {Z | Y ↔ Z ∈ E}) then:
06                 delete S from C_k
07             if S ⊂ X^+ then:
08                 delete S from C_k
09             if U == S^+ then:
10                 delete S from C_k
11 return C_k, Closure;
```

[1] [2] [3] [4]

3 Experimentation

FDTool was initially created to help decompose datasets of medical records as part of the Clinical Archived Records Research for Environmental Studies (CARES) study. The CARES study focuses on 12 datasets obtained from the medical software firms Epic and Legacy. The attribute count in this database ranges from 4 to 18; the row count ranges from 42369 to 8201636.

3.1 Experimental results

FDTool exceeds its time limit (4 hours) and does not terminate when applied to the PatientDemographics dataset (42369 rows $\times$ 18 columns) or the EpicVitals_TobaccoAlcOnly dataset (896962 rows $\times$ 18 columns). The remaining 10 CARES datasets are given in Figure 2.

3.2 Experimental summary

The results from Figure 2 demonstrate that runtime is primarily determined by the number of attributes in a dataset. For instance, the LegacyPayors dataset (1465233 rows $\times$ 4 columns) has more rows and fewer attributes than the AllLabs dataset (1294106 $\times$ 10 columns). The runtime of LegacyPayers (9.4 s.) is much less than that of AllLabs (999.8 s.).

This is because AllLabs has many more edges in its powerset lattice,

$$n \cdot 2^{n-1} - n = 10 \cdot 2^{10-1} - 10 = 5110,$$

Dataset Name	Attribute Count	Row Count	No. of FDs checked	No. of FDs found	No. of Equivalences	Time (s)
AllDxs	7	8201686	112	7	1	577.9
AllLabs	10	1294106	818	43	4	999.8
AllVisits	9	2019117	346	44	7	804.1
EpicMeds	10	1281731	453	26	0	551.9
EpicVitals2016	7	988327	127	2	0	63.0
EpicVitals2015	7	1246303	127	2	0	86.5
FamHx	4	93725	15	0	0	0.7
LegacyIPMeds	8	647122	79	14	1	28.2
LegacyOPMeds	7	740616	33	18	1	7.7
LegacyPayors	4	1465233	15	4	1	9.4
LegacyVitals	8	1453927	146	7	0	134.4

Figure 2: Experimental results of FDTool on 10 CARES datasets.

than does LegacyPayers (28). Hence, FDTool has more FDs to check when applied to AllLabs. It is clear that the atttribute count of a dataset has a much greater effect on FDTool's runtime than does row count.

Many of the edges in the powerset lattice of a candidate are pruned by FDTool. AllLabs has 5110 edges in its powerset lattice. However, only 818 FDs are checked by FDTool, as there are many inferred from the 43 FDs found. This follows from the function *Prune()* deleting many of the candidates to check as a result of mining 4 equivalences. FDTool terminates after 5 k-levels when applied to AllLabs.

4 Operation

FDTool is a command-line Python application executed with the following statement: `$ python fdtool /path/to/file`. This is to be run from the `FDTool` directory. `/path/to/file` is the absolute or relative path to a .txt, .csv, or .pkl file containing a tabular dataset. If the data file has the extension .txt or .csv, FDTool detects the following separators: comma (`','`), bar (`'|'`), semicolon (`';'`), colon (`':'`), and tilde (`'~'`). The data is read in as a Pandas data frame for the algorithm to be carried out with. [5]

FDTool provides the user with minimal FDs and candidate keys mined from a dataset. This is given with the time (s) it took for the algorithm to terminate,

[5]The data is read in with the Pandas function *read_csv()*, which is subject to the usual spacing errors associated with reading in delimiter-separated values.

the row count and attribute count of the data, the number of FDs and equivalences found, and the number of FDs checked. This is printed on the terminal after the code is executed as shown in Figure 3. FDTool writes this information to a .FD_Info.txt file saved to `FDTool/`.

```
MBuranos@LZ2626XMBURANOS MINGW64 ~/Desktop/Python27/Scripts/fdtool (master)
$ python fdtool ../sampleData/Table2.csv

Reading file:
../sampleData/Table2.csv

Functional Dependencies:
{A} -> {D}
{D} -> {A}
{A, B} -> {E}
{B, E} -> {A}
{E, C} -> {A}

Keys:
{B, C, D}
{A, B, C}
{B, C, E}

Time (s): 0.093
Row count: 7
Attribute count: 5
Number of Equivalences: 2
Number of FDs: 5
Number of FDs checked: 19
```

Figure 3: Printed output of FDTool.

4.1 Implementation

FDTool is a Python based re-implemntation of the FD_Mine algorithm. FD_Mine was published in two papers with more detail given to the scientific concepts used in algorithms of its kind [2] [3]. Yao at al. released two versions of FD_Mine with different structures but making use of the same theoretical concepts. Their work is fully justified in mathematical proofs of the pruning rules used [3]. FDTool was coded with special attention given to the pseudo-code in the second version of FD_Mine [3]. [6]

The Python script `dbschema.py` in `FDTool/fdtool/modules/dbschema` is borrowed from *dbschemacmd* (`https://www.elstel.org/database/dbschemacmd.html.en`): a tool for database schema normalization working on functional dependencies. It is used to take well-formatted sets of FDs and infer candidate

[6] FDTool was tested regularly throughout implementation so as to accomodate changes made to improve runtime and memory behavior.

keys from them. This is explained in `FDTool/fdtool/modules/dbschema/Docs`.

4.2 Software availability

1. FDTool is available from the Python Package Index (PyPi; `https://pypi.python.org`) via: pip install fdtool.
2. Latest source code: `https://github.com/mburanosky17/FDTool.git`.
3. License: CC0 1.0 Universal (`https://creativecommons.org/publicdomain/zero/1.0/legalcode`). Module `FDTool/fdtool/modules/dbschema` released under C-FSL license and copyright held by Elmar Stellnberger.
4. Dependencies:
   4.1. Python2 (`https://www.python.org/`), recommended version 2.7.14 or later.
   4.2. Pandas data analysis library (`https://pandas.pydata.org/`).

4.3 Future development

We want to improve its performance so that FDTool is better equipped to handle datasets of different shapes. Using the dependency induction algorithm FDEP, the reach of FDTool could potentially be extended to datasets with more than 100 columns [1]. This might also require upgrading the source code with multicore processing methods, such as a Java API, to reduce runtime and avoid reaching memory limit (100 GB).

Another goal is to increase the functionality provided by FDTool. This would mean implementing the pen and paper methods typically used to normalize relational schema and decompose tables. Our intent is to incorporate these changes in newer versions of FDTool, released at regular periods, so as to develop it as Python software that could automate much of what is done in the database design process.

References

[1] T. Papenbrock, J. Ehrlich, J. Marten, T. Neubert, J.-P. Rudolph, M. Schönberg, J. Zwiener, and F. Naumann, "Functional dependency discovery: An experimental evaluation of seven algorithms," *Proc. VLDB Endow.*, vol. 8, pp. 1082–1093, June 2015.

[2] H. Yao, H. Hamilton, and C. Butz, "Fd_mine: Discovering functional dependencies in a database using equivalences.," 01 2002.

[3] H. Yao and H. J. Hamilton, "Mining functional dependencies from data," *Data Min. Knowl. Discov.*, vol. 16, pp. 197–219, Apr. 2008.

[4] R. Elmasri and S. Navathe, *Database Systems: Models, Languages, Design, and Application Programming.* Pearson, 2011.